# 8

# Pointers

# OBJECTIVES

In this lecture you will learn:

- What **pointers** are.
- The similarities and differences between **pointers** and **references** and when to use each.
- To use **pointers** to pass arguments to functions by reference.

# Introduction

- **Pointers**
  - Powerful, but difficult to master
  - Can be used to perform pass-by-reference
  - Can be used to *create* and *manipulate* dynamic data structures (linked lists, queues, stacks, and trees)
  - *Close relationship* with arrays and strings (`char *` pointer-based strings)

# Pointer Variable Declarations and Initialization

- **Pointer variables**
  - **Contain memory addresses as values**
    - Normally, variable contains specific value (direct reference)
    - Pointers contain address of variable that has specific value (indirect reference)
- **Indirection**
  - **Referencing value through pointer**

# Pointer Variable Declarations and Initialization (Cont.)

- **Pointer declarations**
  - *** indicates variable is a pointer**
    - **Example**
      - `int *myPtr;`
        - Declares pointer to `int`, of type `int *`
        - When `*` appears in decleration; it is not an operator; rather it indicates that the variable being decleared is a pointer
        - Pointers can be decleared to point to objects of any type
    - **Multiple pointers require multiple asterisks**

```
int *myPtr1, *myPtr2;
```

# Pointer Variable Declarations and Initialization (Cont.)

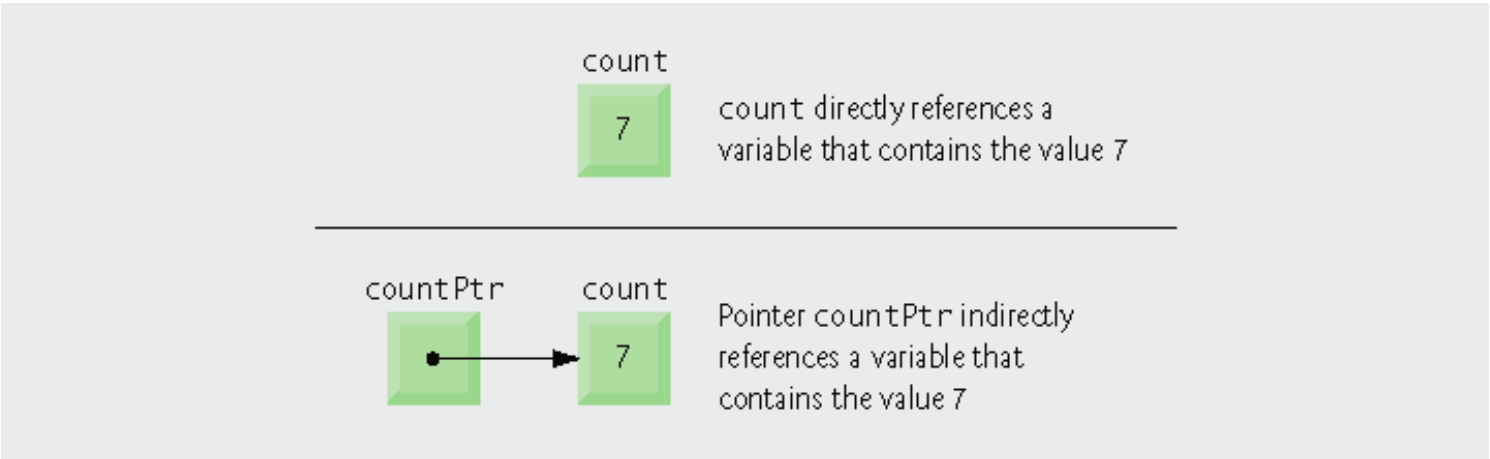
- **Pointer initialization**
  - **Initialized to 0, NULL, or an address**
    - 0 or NULL points to nothing **(null pointer)**
    - The value 0 is the only integer value that can be assigned directly to a pointer variable without casting the integer to a pointer type first.

# Common Programming Error

Assuming that the * used to declare a pointer distributes to all variable names in a declaration's comma-separated list of variables can lead to **errors**. **Each pointer must be declared with the * prefixed to the name** (either with or without a space in between—the compiler ignores the space). Declaring only one variable per declaration helps avoid these types of errors and improves program readability.

# Good Programming Practice

Although it is not a requirement, including the **letters** **`Ptr`** in pointer variable names makes it clear that *these variables are pointers* and that *they must be handled appropriately*.



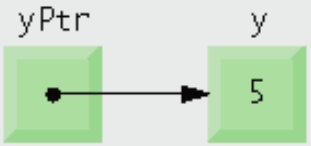

**Directly and indirectly referencing a variable.**

# Error-Prevention Tip

**Initialize pointers** to prevent pointing to unknown or uninitialized areas of memory.

# Pointer Operators

- **Address operator (&)** (Unary operator)
  - **Returns memory address of its operand**
  - **Example**
    - ```
      int y = 5;  //declare variable y
      int *yPtr;  //declare pointer variable yPtr
      yPtr = &y;  //assigns the address of variable y to pointer variable yPtr
      ```
      - **Variable `yPtr` "points to" `y`**
        - `yPtr` indirectly references variable `y`'s value
    - Note that the use of the & in the ***preceding assingnment statement*** is **not the same as** the use of the & in a ***reference decleration***, which is always preceded by data-type-name
  - The **operand** of the address operator must be and **lvalue** (i.e. something to which a value can be assigned, such as variable name, or a reference)
      ```




**Graphical representation of a pointer pointing to a variable in memory.**

# Pointer Operators (Cont.)

- *** operator**
  - Also called **indirection operator** or **dereferencing operator**
  - **Returns synonym** (i.e an alias or a nickname) for the object its operand points to
  - `*yPtr` returns `y` (because `yPtr` points to `y`)
    - `cout << *yptr <<endl;`
    - `cout << y <<endl;`
      - Both statements are equal and they print the value of variable y, namely 5
  - Dereferenced pointer is an ***lvalue***

```
*yptr = 9; //which would assign 9 to y
```

- *** and & are inverses (opposite) of each other**
  - Will "*cancel one another out*" when applied consecutively in either order


**Representation of `y` and `yPtr` in memory.**

# Common Programming Error

**Dereferencing a pointer (**by using * `operator`**)** that has not been properly initialized or that has not been assigned to point to a specific location in memory could cause a **fatal execution-time error**, or it could accidentally modify important data and allow the program to run to completion, possibly with incorrect results.

# Common Programming Error

An attempt to **dereference a variable** that is not a pointer **(**by using `*` `operator`**)** is a **compilation error**.

# Common Programming Error

Dereferencing a null pointer is normally a **fatal execution-time error.**

# Portability Tip

The format in which a **pointer** is output is compiler dependent. Some systems output pointer values as hexadecimal integers, while others use decimal integers.

```
1  // Name Surname, Date, Time.
2  // Using the & and * operators.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int a; // a is an integer
10    int *aPtr; // aPtr is an int * -- pointer to an integer
11
12    a = 7; // assigned 7 to a
13    aPtr = &a; // assign the address of a to aPtr
```

Variable **aPtr** is a pointer to an **int**

Initialize **aPtr** with the address of variable **a**

```
14
15    cout << "The address of a is " << &a
16       << "\nThe value of aPtr is " << aPtr;
17    cout << "\n\nThe value of a is " << a
18       << "\nThe value of *aPtr is " << *aPtr;
19    cout << "\n\nShowing that * and & are inverses of "
20       << "each other.\n&*aPtr = " << &*aPtr
21       << "\n*&aPtr = " << *&aPtr << endl;
22    return 0; // indicates successful termination
23 } // end main
```

Address of **a** and the value of **aPtr** are identical

Value of **a** and the dereferenced **aPtr** are identical

***** and **&** are inverses of each other

```
The address of a is 0012F580
The value of aPtr is 0012F580

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&*aPtr = 0012F580
*&aPtr = 0012F580
```

***** and **&** are inverses; same result when both are applied to **aPtr**

| Operators | Associativity | Type |
|---|---|---|
| `()` `[]` | left to right | highest |
| `++` `--` `static_cast<` *type* `>(` *operand* `)` | left to right | unary (postfix) |
| `++` `--` `+` `-` `!` `&` `*` | right to left | unary (prefix) |
| `*` `/` `%` | left to right | multiplicative |
| `+` `-` | left to right | additive |
| `<<` `>>` | left to right | insertion/extraction |
| `<` `<=` `>` `>=` | left to right | relational |
| `==` `!=` | left to right | equality |
| `&&` | left to right | logical AND |
| `\|\|` | left to right | logical OR |
| `?:` | right to left | conditional |
| `=` `+=` `-=` `*=` `/=` `%=` | right to left | assignment |
| `,` | left to right | comma |

**Operator precedence and associativity.**

# Passing Arguments to Functions by Reference with Pointers

- Three ways to pass arguments to a function
  - **Pass-by-value** (requires passing a copy, original value does not change)
  - **Pass-by-reference with reference arguments**
  - **Pass-by-reference with pointer arguments**
- A function can `return` only one value (or return control from a called function to calling function without passing back a value)
- Arguments passed to a function using reference arguments
  - Function can modify original values of arguments
    - More than one value "returned"

# Passing Arguments to Functions by Reference with Pointers (Cont.)

- Pass-by-reference with pointer arguments
  - Simulates pass-by-reference
    - Use pointers and indirection operator (*)
  - Pass address of argument using & operator
    - When calling a function with an argument that should be modified, the address of the argument is passed.
  - Arrays not passed with & because **array name** is already a pointer
    - The compiler does not differentiate between a function that receives a pointer and a function that receives a one-dimensional array.
    - Name of the array is the starting location in the memory of the array (i.e. The name of an array, `arrayName` , is equvalent to `&arrayName[0].`) (When a compiler encounters a function parameter for a one-dimensional array of the form `int b[ ]`, the compiler converts the parameter to the pointer notation `int *b` )
  - * operator used as alias/nickname for variable inside of function

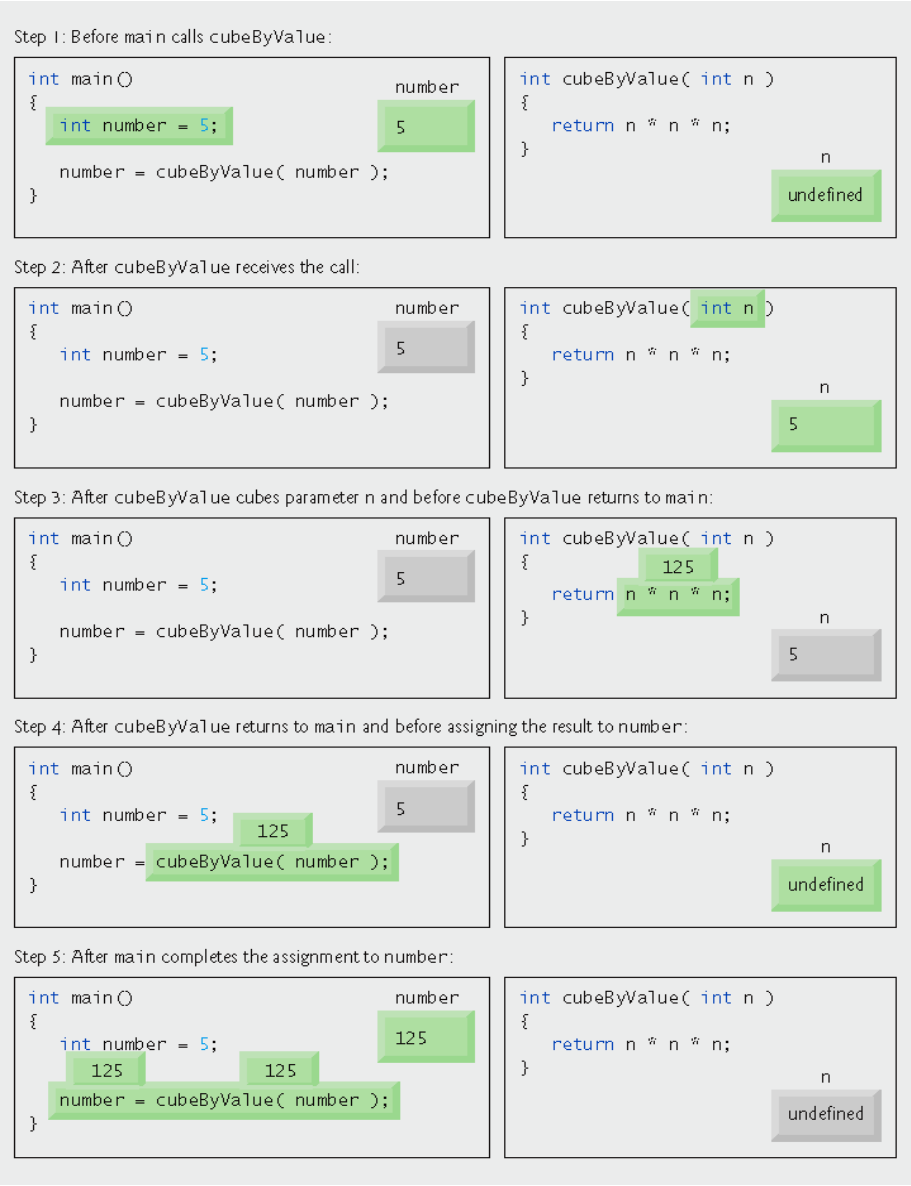
```
1  // Name, Surname, Date, Time
2  // Cube a variable using pass-by-value.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int cubeByValue( int ); // prototype
8
9  int main()
10 {
11    int number = 5;
12
13    cout << "The original value of number is " << number;
14
15    number = cubeByValue( number ); // pass number by value to cubeByValue
16    cout << "\nThe new value of number is " << number << endl;
17    return 0; // indicates successful termination
18 } // end main
19
20 // calculate and return cube of integer argument
21 int cubeByValue( int n )
22 {
23    return n * n * n; // cube local variable n and return result
24 } // end function cubeByValue
```

```
The original value of number is 5
The new value of number is 125
```

Pass number by value; result returned by **cubeByValue**

**cubeByValue** receives parameter passed-by-value

Cubes local variable **n** and **return** the result




**Pass-by-value analysis of the previous program**

# Common Programming Error

Not dereferencing a pointer when it is necessary to do so to obtain the value to which the pointer points is an **error**.

```
1  // Name, Surname, Date, Time.
2  // Cube a variable using pass-by-reference with a pointer argument.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void cubeByReference( int * ); // prototype
8
9  int main()
10 {
11    int number = 5;
12
13    cout << "The original value of number is " << number;
14
15    cubeByReference( &number ); // pass number address to cubeByReference
16
17    cout << "\nThe new value of number is " << number << endl;
18    return 0; // indicates successful termination
19 } // end main
20
21 // calculate cube of *nPtr; modifies variable number in main
22 void cubeByReference( int *nPtr )
23 {
24    *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
25 } // end function cubeByReference
```

```
The original value of number is 5
The new value of number is 125
```

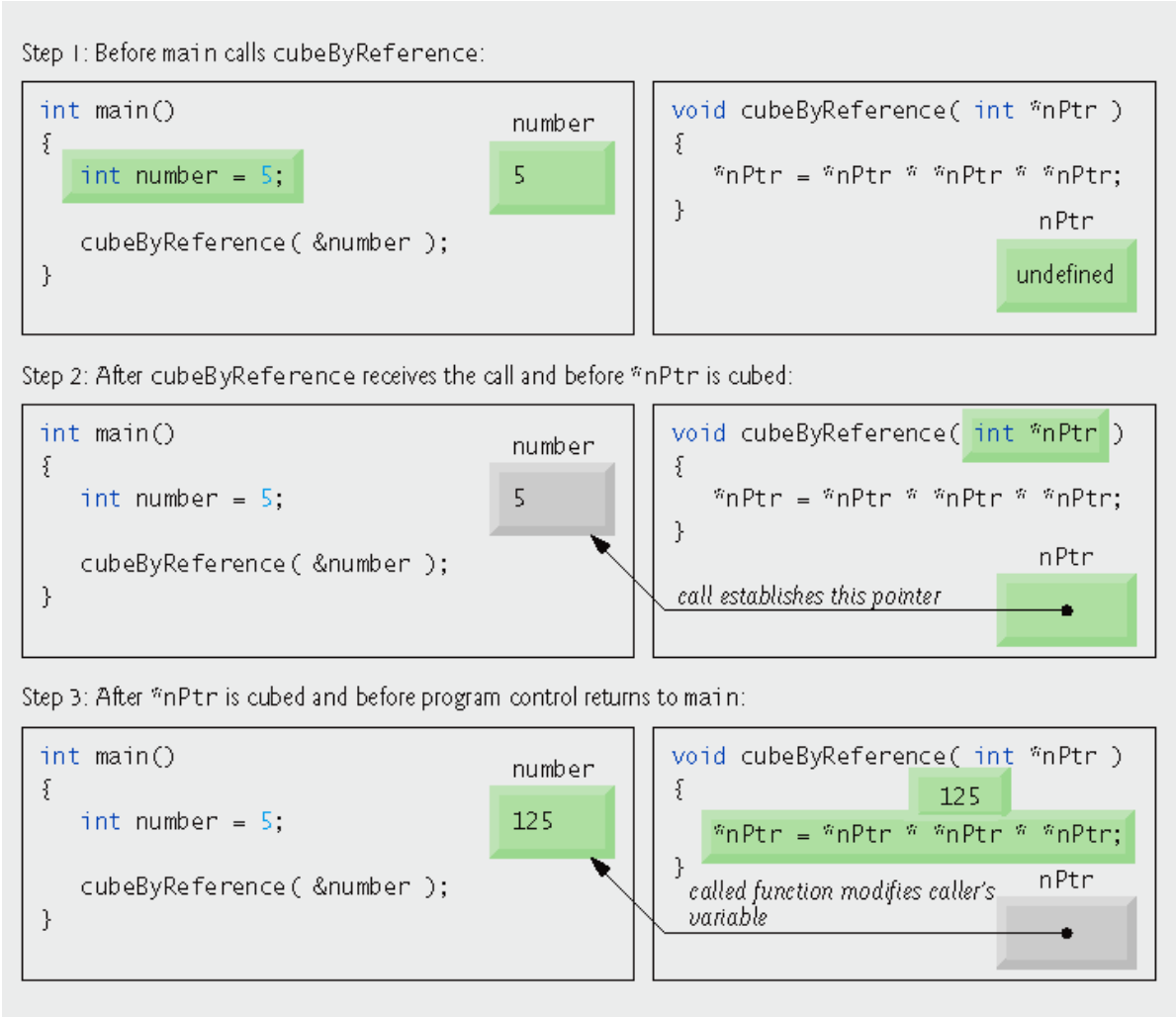
Prototype indicates parameter is a pointer to an **int**

Apply address operator **&** to pass address of **number** to **cubeByReference**

**cubeByReference** modifies variable **number**

Modify and access **int** variable using indirection operator ***

**cubeByReference** receives address of an **int** variable, i.e., a pointer to an **int**




**Pass-by-reference analysis (with a pointer argument) of the previous program.**

# Software Engineering Observation

Use **pass-by-value** to pass arguments to a function unless the caller explicitly requires that the called function directly modify the value of the argument variable in the caller. This is another example of the principle of least privilege.

# Using `const` with Pointers

- **`const` qualifier**
  - Indicates that value of variable should not be modified
  - `const` used when function does not need to change the variable's value
- **Principle of least privilege**
  - Award function enough access to accomplish task, but no more
  - Example
    - A function that prints the elements of an array, takes array and `int` indicating length
      - Array length is not changed – should be **`const`**
      - Array contents are not changed – should be **`const`**
        - This is especially important because an entire array is *always* passed by reference and could easily be changed in the called function !

# Portability Tip

Although **`const`** is well defined in ANSI C and C++, some compilers do not enforce it properly. So a good rule is, "Know your compiler."

# Software Engineering Observation

If a value does not (or should not) change in the body of a function to which it is passed, the parameter should be declared **`const`** to ensure that it is not accidentally modified.

# Error-Prevention Tip

Before using a function, check its function prototype to determine the parameters that it can modify.

# Using `const` with Pointers (Cont.)

- **Four ways to pass pointer to function**
  - **Nonconstant pointer to nonconstant data**
    - Highest amount of access
    - Data can be modified through the dereferenced pointer
    - Pointer can be modified to point to other data
      - Pointer arithmetic
        - Operator ++ moves array pointer to the next element
    - Its declaration does not include **`const`** qualifier

```
1  // Name Surname, Date, Time.
2  // Converting lowercase letters to uppercase letters
3  // using a non-constant pointer to non-constant data.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include <cctype> // prototypes for islower and toupper
9  using std::islower;
10 using std::toupper;
11
12 void convertToUppercase( char * );
13
14 int main()
15 {
16    char phrase[] = "characters and $32.98";
17
18    cout << "The phrase before conversion is: " << phrase;
19    convertToUppercase( phrase );
20    cout << "\nThe phrase after conversion is:  " << phrase << endl;
21    return 0; // indicates successful termination
22 } // end main
```

Parameter is a nonconstant pointer to nonconstant data

**convertToUppercase** modifies variable **phrase**

```
23
24 // convert string to uppercase letters
25 void convertToUppercase( char *sPtr )
26 {
27    while ( *sPtr != '\0' ) // loop while current character is not '\0'
28    {
29       if ( islower( *sPtr ) ) // if character is lowercase,
30          *sPtr = toupper( *sPtr ); // convert to uppercase
31
32       sPtr++; // move sPtr to next character in string
33    } // end while
34 } // end function convertToUppercase
```

```
The phrase before conversion is: characters and $32.98
The phrase after conversion is:  CHARACTERS AND $32.98
```

Parameter **sPtr** is a nonconstant pointer to nonconstant data

A character array's name is really equivalent to a pointer to the first character of the array, so passing **phrase** as an argument to **convertToUppercase** is possible.

Function **islower** returns **true** if the character is lowercase

Function **toupper** returns corresponding uppercase character if original character is lowercase; otherwise **toupper** returns the original character

Modify the memory address stored in **sPtr** to point to the next element of the array. This would not be possible if **sPtr** were declared **const.**

# Using const with Pointers (Cont.)

- **Four ways to pass pointer to function (Cont.)**
  - **Nonconstant pointer to constant data**
    - Pointer can be modified to point to any data item of the appropriate type
    - However, the data to which it points cannot be modified through this pointer
    - Provides the performance of pass-by-reference and the protection of pass-by-value

```
1  // Name Surname, Date, Time.
2  // Printing a string one character at a time using
3  // a non-constant pointer to constant data.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  void printCharacters( const char * ); // print using pointer to const data
9
10 int main()
11 {
12    const char phrase[] = "print characters of a string";
13
14    cout << "The string is:\n";
15    printCharacters( phrase ); // print characters in phrase
16    cout << endl;
17    return 0; // indicates successful termination
18 } // end main
19
20 // sPtr can be modified, but it cannot modify the character to which
21 // it points, i.e., sPtr is a "read-only" pointer
22 void printCharacters( const char *sPtr )
23 {
24    for ( ; *sPtr != '\0'; sPtr++ ) // no initialization
25       cout << *sPtr; // display character without modification
26 } // end function printCharacters
```

```
The string is:
print characters of a string
```

Parameter is a nonconstant pointer to constant data

Pass pointer **phrase** to function **printCharacters**

**sPtr** is a nonconstant pointer to constant data; it cannot modify the character to which it points

Increment **sPtr** to point to the next character

This works because (the pointer) **sPtr** is not **const.**

```
1  // Name Surname, Date, Time.
2  // Attempting to modify data through a
3  // non-constant pointer to constant data.
4
5  void f( const int * ); // prototype
6
7  int main()
8  {
9     int y;
10
11    f( &y ); // f attempts illegal modification
12    return 0; // indicates successful termination
13 } // end main
```

Parameter is a nonconstant pointer to constant data

Pass the address of **`int`** variable **`y`** to attempt an illegal modification

```
14
15 // xPtr cannot modify the value of constant variable to which it points
16 void f( const int *xPtr )
17 {
18    *xPtr = 100; // error: cannot modify a const object
19 } // end function f
```

Attempt to modify a **`const`** object pointed to by **`xPtr`**

***Borland C++ command-line compiler error message:***

```
Error E2024 fig08_12.cpp 18:
   Cannot modify a const object in function f(const int *)
```

***Microsoft Visual C++ compiler error message:***

```
c:\cpphtp5_examples\ch08\Fig08_12\fig08_12.cpp(18) :
   error C2166: l-value specifies const object
```

Error produced when attempting to compile

***GNU C++ compiler error message:***

```
fig08_12.cpp: In function `void f(const int*)':
fig08_12.cpp:18: error: assignment of read-only location
```

# Performance Tip

- When a function is called with an **array** as an argument, the array is passed to function by reference. However, **objects** are always passed by value - a copy of entire object is passed. (This requires the execution-time overhead of making a copy of each data item in the object and storing it on the function call stack.)
- If they do not need to be modified by the called function, pass **large objects** using pointers to constant data or references to constant data, to obtain the *performance* benefits of pass-by-reference.

# Software Engineering Observation

•Pass large objects using pointers to constant data, or references to constant data, to obtain the *security* of pass-by-value.

# Using `const` with Pointers (Cont.)

- **Four ways to pass pointer to function (Cont.)**
  - **Constant pointer to nonconstant data**
    - Always points to the same memory location
      - Can only access other elements using subscript notation
    - However, data at that location can be modified through the pointer
    - This is the default for an array name (An array name is a constant pointer to *the beginning of the array*. *All data* in the array can be *accessed* and *changed* by using the array name and array subscripting. A constant pointer to nonconstant data can be used to receive an array as an argument to a function that access array elements using array subcripting notation)
      - Can be used by a function to receive an array argument
    - Pointers that are declared `const` must be initialized when declared (If the pointer is a function parameter, it is initialized with a pointer that is passed to the function)

```
1  // Name Surname, Date, Time.
2  // Attempting to modify a constant pointer to non-constant data.
3
4  int main()
5  {
6     int x, y;
7
8     // ptr is a constant pointer to an integer that can
9     // be modified through ptr, but ptr always points to the
10    // same memory location.
11    int * const ptr = &x; // const pointer must be initialized
12
13    *ptr = 7; // allowed: *ptr is not const
14    ptr = &y; // error: ptr is const; cannot assign to it a new address
15    return 0; // indicates successful termination
16 } // end main
```

**ptr** is a constant pointer to an integer

Can modify **x** (pointed to by **ptr**) since **x** is not constant

Cannot modify **ptr** to point to a new address since **ptr** is constant

*Borland C++ command-line compiler error message:*

```
Error E2024 fig08_13.cpp 14: Cannot modify a const object in function main()s
```

*Microsoft Visual C++ compiler error message:*

```
c:\cpphtp5e_examples\ch08\Fig08_13\fig08_13.cpp(14) : error C2166:
   l-value specifies const object
```

*GNU C++ compiler error message:*

```
fig08_13.cpp: In function `int main()':
fig08_13.cpp:14: error: assignment of read-only variable `ptr'
```

Line 14 generates a compiler error by attempting to assign a new address to a constant pointer

# Common Programming Error

Not initializing a pointer that is declared `const` is a **compilation error**.

# Using const with Pointers (Cont.)

- **Four ways to pass pointer to function (Cont.)**
  - **Constant pointer to constant data**
    - *Least amount of access*
    - Always points to the same memory location
    - Data at that memory location cannot be modified using this pointer
    - This is how an array should be passed to a function that only reads the array, using array subscript notation, and does not modify the array

```
1  // Name Surname, Date, Time.
2  // Attempting to modify a constant pointer to constant data.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int x = 5, y;
10
11    // ptr is a constant pointer to a constant integer.
12    // ptr always points to the same location; the integer
13    // at that location cannot be modified.
14    const int *const ptr = &x;
15
16    cout << *ptr << endl;
17
18    *ptr = 7; // error: *ptr is const; cannot assign new value
19    ptr = &y; // error: ptr is const; cannot assign new address
20    return 0; // indicates successful termination
21 } // end main
```

This decleration is read from right to left as "**ptr** is a constant pointer to an integer constant"

**ptr** is a constant pointer to a constant integer

Cannot modify **x** (pointed to by **ptr**) since ***ptr** is constant

Cannot modify **ptr** to point to a new address since **ptr** is constant

*Borland C++ command-line compiler error message:*

```
Error E2024 fig08_14.cpp 18: Cannot modify a const object in function main()
Error E2024 fig08_14.cpp 19: Cannot modify a const object in function main()
```

*Microsoft Visual C++ compiler error message:*

```
c:\cpphtp5e_examples\ch08\Fig08_14\fig08_14.cpp(18) : error C2166:
   l-value specifies const object
c:\cpphtp5e_examples\ch08\Fig08_14\fig08_14.cpp(19) : error C2166:
   l-value specifies const object
```

*GNU C++ compiler error message:*

```
fig08_14.cpp: In function `int main()':
fig08_14.cpp:18: error: assignment of read-only location
fig08_14.cpp:19: error: assignment of read-only variable `ptr'
```

Line 18 generates a compiler error by attempting to modify a constant object

Line 19 generates a compiler error by attempting to assign a new address to a constant pointer

# Selection Sort Using Pass-by-Reference

- **Implement `selectionSort` using pointers**
  - **Selection sort algorithm**
    - Swap smallest element with the first element
    - Swap second-smallest element with the second element
    - Etc.
  - **Want function `swap` to access array elements**
    - Individual array elements: scalars
      - Passed by value by default (Although entire arrays are passed by reference, individual array elements are scalars and are ordinarily passed by value)
    - Pass by reference via pointers using address operator &

```
1  // Name Surname, Date, Time.
2  // This program puts values into an array, sorts the values into
3  // ascending order and prints the resulting array.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9  using std::setw;
10
11 void selectionSort( int * const, const int ); // prototype
12 void swap( int * const, int * const ); // prototype
13
14 int main()
15 {
16    const int arraySize = 10;
17    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
18
19    cout << "Data items in original order\n";
20
21    for ( int i = 0; i < arraySize; i++ )
22       cout << setw( 4 ) << a[ i ];
23
24    selectionSort( a, arraySize ); // sort the array
25
26    cout << "\nData items in ascending order\n";
27
28    for ( int j = 0; j < arraySize; j++ )
29       cout << setw( 4 ) << a[ j ];
```

```
30
31    cout << endl;
32    return 0; // indicates successful termination
33 } // end main
34
35 // function to sort an array
36 void selectionSort( int * const array, const int size )
37 {
38    int smallest; // index of smallest element
39
40    // loop over size - 1 elements
41    for ( int i = 0; i < size - 1; i++ )
42    {
43       smallest = i; // first index of remaining array
44
45       // loop to find index of smallest element
46       for ( int index = i + 1; index < size; index++ )
47
48          if ( array[ index ] < array[ smallest ] )
49             smallest = index;
50
51       swap( &array[ i ], &array[ smallest ] );
52    } // end if
53 } // end function selectionSort
```

Declare **`array`** as **`int *array`** (rather than **`int array[]`**) to indicate function **`selectionSort`** receives single-subscripted (one-dimentional) array as an argument

When a pointer-based array is passed to a function, only the memory address of the first element of the array is received by the function; array size must be passed separately to the function.

Receives the size of the array as an argument (because the function must have that information to sort the array); declared **`const`** to ensure that **`size`** is not modified

```
54
55 // swap values at memory locations to which
56 // element1Ptr and element2Ptr point
57 void swap( int * const element1Ptr, int * const element2Ptr )
58 {
59    int hold = *element1Ptr;
60    *element1Ptr = *element2Ptr;
61    *element2Ptr = hold;
62 } // end function swap
```

Arguments are assed by reference, allowing the function to swap values at the original memory locations

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

# Software Engineering Observation

When passing an array *to a function*, also pass the size of the array (rather than building into the function knowledge of the array size). This makes the function more **reusable**.